`lfd_infer`: an Implementation of a Static Inference on Heap Space Usage

Steffen Jost
LMU München, Institut für Informatik, Oettingenstraße 67, 80538 München, Germany
jost@informatik.uni-muenchen.de

ABSTRACT

We will exhibit an implementation of the static prediction of heap space usage for first-order functional programs as proposed by M. Hofmann and S. Jost in 2003.

The implementation shows that this static inference can be implemented efficiently and is able to identify upper bounds on the consumption of heap memory for a large range of example programs. We will use these examples to investigate the inference's strengths and weaknesses.

Furthermore, this implementation extends the theoretic work by including annotation-related subtyping and arbitrary recursive data types.

1. INTRODUCTION

In [7], M. Hofmann and S. Jost presented a static inference for heap space bounds on first-order functional programs. We shall quickly recall the basic ideas of this inference by considering a small program example.

Let us consider the heap space usage of the well known Insertionsort algorithm written in the experimental language Camelot [19], an ML-dialect with memory primitives, as shown in figure 1. We will comment more on Camelot soon in section 1.3.

```
let insert e l =
  match l with [] -> e::[]
  | (h::t)@_      -> if h >= e then e::(h::t)
                                else h::(insert e t)
let sort l =
  match l with [] -> [] | h::t -> insert h (sort t)
```

Figure 1: Insertionsort

The function `sort` takes a list of integers and inserts the head `h` of the list into the recursively sorted tail `t` at the right place by using `insert`. This function takes an integer `e` and a (presumably sorted) list `l` and traverses `l` until it reaches an element `h` of `l` that is bigger than or equal to `e`. In this case it inserts `e` before `h` in the list.

We are interested in the heap space consumption of this program, so we must fix a memory model first: Assume integers are unboxed; while a list node consisting of an integer and a pointer to the next node of the list consumes an atomic portion of heap space that we will refer to as a heap cell. Furthermore assume that the empty list is represented by a nil pointer and therefore does not need any heap space for storing.

We could have assumed that a list node rather requires two heap cells for storing, one for the pointer and one for the integer. In fact, any nonnegative number of heap cells is acceptable, depending solely on the desired memory model. Note that the memory model is not inherent to the inference, therefore we speak in terms of heap cells rather than an absolute measure. The size of a heap cell determines our finest grain of measurement. Applying the inference to the same program using different memory models will produce different results, of course.

Function `insert` has three paths of computation: The first, when `l` is empty, uses up one heap cell for storing `e` on top of an empty list. The second path, the `then`-branch, clearly uses two heap cells for storing `h` and then `e` on to of `t`, while the third one, the `else`-branch, uses up one heap cell again.

Although we do not want `sort` to deallocate its input, it is reasonable to deallocate the interim sorted partial lists. Hence we allow `insert` to deallocate the heap cell where `h` was stored in, which we signify by writing `@_` in the corresponding match branch. Thence this heap cell is now available for use in one of the subsequent storing operations. Note that we merely care for the amount of heap cells available, so we assume that deallocated heap cells are remembered in a freelist or by another suitable mechanism.

So now all branches of computation of `insert` only consume one heap cell, except for the `else`-branch which does not use up heap memory at all, but since `insert` calls itself recursively in this branch, we deduce that every ('initial') call to `insert` uses up precisely one heap cell. We record this information, that a call to function `insert` consumes *one* heap cell and returns *zero* unused heap cells, within the type of the function by writing: `insert: 1, int -> list -> list, 0;`

Now we have to consider the heap space consumption of `sort`. We could write `@_` in its pattern match as well, and pass the thus gained heap cell on to `insert` as a special argument, but as mentioned before, for a reason which eludes

```
insert: 1, int -> list[Cons(int,#,0)|Nil(0)]
        -> list[Cons(int,#,0)|Nil(0)], 0;

sort  : 0, list[Cons(int,#,1)|Nil(0)]
        -> list[Cons(int,#,0)|Nil(0)], 0;
```

The occurring `1` in the type of `sort` tells us that executing `sort` may allocate a number of fresh heap cells equal to the length of its input list, i.e. *one* heap cell per `Cons`-node of the input list.

The `1` in the type of `insert` tells us that a call to `insert` may allocate one heap cell during its execution (including all possible subcalls).

For brevity, we might sometimes omit printing the name of each constructor within an annotated type, and print only the annotated types of its arguments, e.g. we write `list[int,#,7|8]` instead of `list[Cons(int,#,7)|Nil(8)]`.

Figure 2: Enriched signature for Insertionsort

our scope, `sort` should preserve its input list. So all we can deduce is that for each node of the input list there should be at least one heap cell available in order to prevent the evaluation of getting stuck due to a lack of heap space, since we call `insert` exactly once for each node of `l`.

Again we record this information within the type of `sort`, but this time it is a little more complicated if we want to use our method for arbitrary types later on. For each occurring type, we must list all its constructors and for each constructor we add a number indicating the amount of heap cells that must be supplied for each node contained in the input of that type. The resulting signature is shown in figure 2. We will refer to such a typing as an *enriched* or *annotated* typing.

Note that in order to account for nested datatypes, we list the annotated types of all arguments for each constructor as well. We always write the number of additional heap cells carried by a particular constructor at the end of the listing of its arguments. Within the argument listing, the symbol `#` stands for the constructors enclosing annotated type, i.e. the type itself with exact the same resource annotations.

So each internal node of an input list for `sort` must contain an integer, a list of the same annotated type (i.e. the tail) and a voucher for one unused heap cell. An internal node of an input list for `insert` only contains an integer and the tail (again of the same annotated type), but does not bring a voucher for an available heap cell along. Note that these "vouchers" are imaginary, i.e. there is no runtime overhead. Both nodes are identical at runtime, they are just treated differently in our analysis.

One should also note that for reasons of modularity, the type of the output of a function carries resource annotations as well. In fact, the amount of unused heap cells available after the computation of a function is given in terms of the result itself and its type, rather than being related to input of a function. We show an example of the overall memory picture shortly within the next section.

1.1 Burdens...

First of all, we admit that forcing the programmer to designate for each pattern match whether it is destructive or not is a great burden. Indeed, as pointed out already by ourselves and several times again by fellow researchers, our inference heavily relies upon a guarantee that the programmer chooses the right pattern match mode so that no dangling pointers are dereferenced in any run. Therefore a safe and static automation of these decisions is highly desirable.

Static systems to alleviate these problems are an issue in ongoing research, yet there is already some notable scientific research available [15, 11, 18, 3, 12]. Since the problem was well isolated in our work, one is free to choose among the proposed systems for ensuring the safety of the inferred heap space bounds while using the inference as it is now.

As an alternative one could also rely on a linear typing discipline, which would also ensure referential transparency, as all pattern matches can then be safely set to be destructive. The expressiveness of such systems still covers many examples and is well identified [6]. However, from a practical point of view, writing linearly typeable programs seems overly restrictive.

All other requirements on the treated programs, namely monomorphism, sequentiality (let-normal form) and unique identifiers may be obtained automatically by precompiling with the Camelot compiler.

1.2 ... and benefits

This implementation works as follows:

- We extract a set of linear inequalities over the rationals on the fly by reconstructing a typing derivation for a given program. This was described in detail in [7], so we will only treat examples here.
- These inequalities, together with a rather arbitrary objective function, form a linear program (LP), which is fed to an external LP-solver.
- The solution computed by the LP-solver is then used to establish an annotated typing for each of the program's functions, which yields a linear bound on the function's heap space consumption.

We will describe the interpretation of the inferred annotated signatures with an example now: If for a function `f` of type

```
type list = Nil(*0*) | Cons(*1*) of int * list
val f : list -> list -> list
```

the inference computes the annotated type

```
val f : 2, ilist[Nil(3)|Cons(int,#,4.5)]
        -> ilist[Nil(6)|Cons(int,#,7)]
        -> ilist[Nil(0)|Cons(int,#,8)], 0;
```

(where `#` refers to the constructor's annotated type itself), then we see that computing `f(a,b)` on two integer lists `a` and `b` with, say, length 22 and 33 respectively, will require $2 + \big(3 + (4.5 \cdot 22)\big) + \big(6 + (7 \cdot 33)\big) = 341$ *additional* free heap cells. Please note the calculation and the position of the numbers within: list `a` consists of 22 `Cons`-nodes and one `Nil`-Node. According to the required type of `a`, there must be 4.5 additional heap cells available for allocation for each `Cons`-node of `a` and 3 for each `Nil`-node, hence $(3+(4.5\cdot 22)$. The third addend in the above calculation corresponds the list 'b'.

If the computation would result in an integer list of length, say, 44, we then knew by the annotated result type of `f`, that at least $\big(0+(8\cdot 44)\big)+0 = 352$ heap cells were available again after computing `f(a,b)`. These heap cells might or

might not have been used during the computation, but it is guaranteed that they do not contain live data after the computation of `f` is finished.

The meaning of the notion *heap cell* entirely depends on our desired memory model. In the above example, we stated within the comments behind the constructor definitions that each node of type list requires 1 heap cell for storing, while the list `Nil` will occupy no heap space at all. In general, each constructor in a type declaration should carry such an integer comment after its name to denote the number of heap cells required to store a node of this constructor on the heap in the desired memory model.

So in the above example, $(1 \cdot 22) + (1 \cdot 33) = 55$ heap cells were allocated to store `a` and `b` before the call, hence a total of $55 + 341 = 396$ heap cells were at most involved for computing `f(a,b)`. Likewise, the result itself would occupy 44 heap cells, so together with the heap cells that are guaranteed to be free again after the computation, we obtain $44 + 352 = 396$ in total again, and see in this way that no memory was leaked by computing `f(a,b)`.

However, if the result would be a list of length 42 only, our calculation of the total after the computation would be $1 \cdot 42 + 8 \cdot 42 = 378$, hence $396 - 378 = 18$ heap cells would have been leaked. We also know that a result of length 45 or longer is not possible with the input `a` and `b`, just by glancing at the annotated type of `f`.

All our inference provides is a strict linear upper bound on the heap space consumption relative to a function's weighted input sizes, if such a bound is derivable in our system. We will study a more realistic example demonstrating memory leakage in section 3.3. Note that memory leakage will always show in strict inequalities of the solution obtained for the constructed LP during the inference, so in case that a solution to the LP exists at all, memory leakage could be traced back to the source as explained in section 3.8.

In addition, this implementation of the heap-space inference is equipped with a facility for conducting sandboxed execution runs on its own, thereby speeding up testing and allowing toying around in order to gain more insight in the inference's results. We will refer to this implementation in the following as `lfd_infer`. It is available for download at the author's webpage [9].

1.3 The Camelot compiler

As stated above, using the Camelot compiler [19] for precompiling may provide some significant convenience for producing programs digestible by `lfd_infer`. In addition, a non-well typed program might sometimes still lead to an annotated type, which is then likely to be erroneous. We therefore rely on the Camelot compiler for type checking programs prior to inference as well.

Camelot is currently developed within the EU-funded project *Mobile Resource Guarantees* [1], which aims at developing a system for generating portable proof-carrying code entailing safety aspects of mobile programs such as time and space.

Camelot compiles an ML-style language into a subset of the Java virtual machine called Grail, for which a system to write machine-checkable proofs is developed.

Since our implementation of the heap space inference relies on Camelot for monomorphisation, let-normal from and type checking, the language it recognizes is heavily related to Camelot. Vice versa, programs recognized by this inference should require at most minor modifications in order to be accepted by the Camelot compiler.

1.4 Inferring integer annotations

It was shown in [7] that integer annotations are of special interest, as they enable us to translate a linearly typed program into `malloc()`-free C code as described in [6]. Furthermore we already proved in that work that integer annotations can be computed within polynomial time.

Although `lfd_infer` has an option to restrict the inferred annotations to integers rather than rationals, we make no use of the methods described in [7] to compute the annotation within polynomial time, as it turned out that the occurring LPs produced by the inference behave well enough to allow the used LP-solver to efficiently compute an integer solution directly on its own.

For now we are contend that our theory still yields room for improving the efficency of the inference, if efficiency of computing integer annotations ever becomes a concern in the future.

2. THEORY & PRACTICE

As one would expect, the implementation of the inference differs from the theoretical work in a number of ways: First of all, lists, pairs or sum types are not built-in. The user is allowed to define *arbitrary recursive data types*,* including the aforementioned ones. All built-in types are assumed to be unboxed (i.e. heap free), including the string type. This is of course unnatural, but strings were only included for the purpose of simple screen prints in toy executions of the analyzed programs. If heap space consumption of strings is a concern, these must be treated as proper lists of type `char`. Alternatively one might define a datatype of fixed length strings.

As explained in section 1, each constructor is assigned a size. (This size directly corresponds to the $\mathsf{SIZE}()$-function found in the introduction and elimination rules for lists in $\mathsf{LF}_\diamond$ in [7].) This leads to some ambiguity, e.g. fix some integer c, then there is no real difference between all types obtained by instantiating the type scheme x`, ilist[Nil(`y`)|Cons(int,#,0)] -> unit, 0` so that $x + y = c$, since there is precisely one `Nil` node in every list. The only difference in the case of lists would be that the function must traverse (either read-only or destructively) its input list entirely before using those y heap cells, while the x heap cells would be right available from the start, although traversing the list would change nothing to the heap usage at all. So the inference will naturally prefer x over y, and it will not matter otherwise.

One can make up more examples with other data structures to exploit more ambiguities with resource annotations, but this is not a real problem, as it only expands the set of solutions to the LP. The author discussed these matters already in [10] extensively.

In addition, and for compatibility with Camelot, the syntax for distinguishing a destructive- from a read-only pattern match was changed slightly. Now each single branch of a

*Except for mutual recursive data types, which currently lead to non-termination. This is not a problem of the inference mechanism, but is merely due to the programmer's inexperience in dealing with `#`. There are well-known techniques to deal with this problem, but currently this matter is simply not pressing enough.

pattern match is marked as either read-only or destructive, allowing more flexibility but requiring much more typing rules in theoretical work, which are all almost identical and straightforward to produce. Also the type scheme for the sharing (or contraction) rules must be expanded to include each new type. The syntax is now as follows:

```
match x with
  | Constr(a,b,c)   -> ... (* Read-only match   *)
  | Constr(a,b,c)@_ -> ... (* Destructive match *)
```

Of course, for each constructor only one of the above possibilities is allowed in a single match operation.

Subtyping with respect to resource annotations may be allowed when using `lfd_infer`. This naturally extends the idea of the implicit typing rule $\mathsf{LF}_\Diamond$:Waste. We will comment more on this in example 3.7.

Since modularity is inherent to the inference, `lfd_infer` supports *partial inference*. A function which is not declared (i.e. there is no `val`-statement) is simply ignored by the inference. The inference will fail if this function is called by any other declared and defined function of the program.

On the contrary, we might desire to *enforce* some *annotations*, e.g. for external functions. This can be done using `richval` instead of `val` and then stating the function's annotated type as it would be printed[†] by `lfd_infer`. So one could write

```
richval g:2,list[Nil(0)|Cons(int,#,3.4)] -> unit,*;
```

to enforce that `g` requires 2 fixed heap cells and 3.4 per `Cons`-node of its input list when called. However, we still may want to leave some values to the inference to decide, which we denote by `*`.

Any defined function is then checked against the given declaration. If a function is left undefined, all unknown annotations are assumed to be zero and otherwise accepted as given.

If the LP solver fails to solve the LP for the program at once, i.e. there is no valid annotated signature. In this case, `lfd_infer` considers each function on its own to help identifying why no linear bound on the heap space usage of the program could be inferred. Usually the problem can be identified within a certain function of the program in this way. But it might also be the case that although there is a solution to each function's particular set of constraints, there is no solution to the whole LP of the program, e.g. mutual recursive functions which lower (or consume) resources brought by some input list and passing this list on to each other will produce such a result. This is however as it should be, as combining these functions does indeed lead to super-linear heap space consumption.

3. EXAMPLES

In the following section we will look at various examples, discuss the meaning of the results, the efficiency for obtaining them, and other various benefits or problems encountered.

[†]The syntax is slightly different than shown, as `lfd_infer` knows two printing styles: a verbose style (default) and a condensed style (option `-ndia`). A `richval` declaration requires the verbose style, but to avoid confusion, we are only using the condensed style throughout this paper.

3.1 Efficiency

All measurements were done on a Mobile Intel Pentium 4 Notebook with 1,8GHz and 256MB RAM available. Notice that we could not disable CPU speed reduction under Linux, so the processor might have actually been somewhat slower.

`lfd_infer`'s own total time printout corresponds to the GNU time utilities user-time measure, but was measured with time routines provided by OCaml. Further detailed measurements of the inference's core routines are only printed to the screen if they are greater than 0.03s. For small programs, the most amount of the total processing time is indeed used for lexing and parsing using standard tools and reading or writing to the file system.

We are considering now a series of examples, called `big`X`.lfd`, which essentially consist of X functions which all behave like the identity function on integer lists and are identical up to renaming of identifiers. Each function consists of 6 operations: a pattern match, calling itself once and its predecessor twice, creating an empty list and adding an element to a list. So the program `big2000.lfd` contains more than 12000 instructions within its 24188 lines of code. An excerpt of the program `big100.lfd` is given in appendix A.1. We observe:

#fun	**#ieqs**	**#vars**	**Tcnst**	**Tsolv**	**TT**
100	2117	1408	0.01s	0.31s	0.34s
200	4117	2708	0.04s	1.04s	1.13s
500	10117	6608	0.09s	9.02s	9.22s
1000	20117	13108	0.18s	47.18s	47.62s
2000	40117	26108	0.37s	201.42s	202.34s

The two last calls, having 48s and 203s user time (**TT**), corresponded to less than 3 and 17 minutes real time respectively. User time is the number of CPU-seconds the program was actively running; while real time is the elapsed time as perceived by a waiting human. Real time depends on the IO performance of the operating system, the amount of processes running, etc.

Not surprisingly, the number of inequalities (**#ieqs**) and variables (**#vars**) of the constructed LPs is linear in the number of functions (**#fun**) of the program. This is due to the inferences modularity, as each function is examined independently! Hence it does not matter to the inference that all examined functions are essentially identical. For each function, the 20 generated inequalities are identical up to renaming of variables. 13 fresh variables are introduced with each function. This is nicely reflected in the linear runtime required to generate each LP (**Tcnst**). The remaining inequalties and variables belong to the main function.

However, solving the generated LP is much more complicated as all inequalities are intertwined by the variables (due to the interweaved function calls). So the kind of operations barely affects the size of the generated LP, but it does affect the LP's complexity.

Tsolv refers to the time required by the tool `lp_solve` [4] to compute a solution to the generated LP; it is called via Unix pipe. **TT** refers to the total user time of the computation. Note that **TT** heavily depends upon whether the constructed LP is also written to a file or merely passed over the pipe to the LP solver. The times given in the table above were obtained by using options `-lp` and `-perf`, which suppresses writing of the LP to an unnecessary protocol file. `-perf` also prevents sorting and labeling of the generated inequalities and might lead to a wrong count of inequalities, **#ieqs** has thus been corrected (Trival inequalities are never

passed to the solver, but are counted under this option. In the above example, four trivial inequalties are produced per function).

The required computation times seem quite acceptable, especially since using a commercial LP solver instead might further improve these results. In addition, `lp_solve` was called without any options, which perhaps could improve performance on the supplied kind of LP.

Since for *all* other examples that we have examined so far the annotated signature is inferred within less than 0.05s user time or barely 1s real time, we will not consider computation time for the remainder of this work anymore.

3.2 Heapsort - a complete example run

In order to give an impression how the inference works, we will consider a complete example run now. The program `heapsort.cmlt`, which implements the Heapsort algorithm, is given in A.2. After processing the file with Camelot we obtain the file `heapsort.lfd`, which we simply feed into the inference as shown in figure 3.

The annotated signature of function `sort` shows that it may be executed with no additional heap space at all. Of course, this is done by in-place sorting. If the input is to be preserved, the signature would become

```
sort: 0, list_2[int,#,1|0] -> list_2[int,#,0|0], 0;
```

in a memory model where each list node occupies one heap cell, thus demanding the same amount of extra heap memory as the input itself.

Nevertheless, this might still be surprising, as `sort` implements the well-known Heapsort algorithm, and one might expect that a heap-tree occupies more memory resources than a simple list: Yet this is not true in this case, as Camelot's memory model assumes that each heap cell is big enough to contain any node of any occurring data structure. If we want to consider a different memory model, for example we might want that an internal tree node requires 3 heap cells, while an internal list node occupies 2 cells within the heap, we have change the numbers given in the data declarations within the file `heapsort.lfd`. Hence we write

```
type tree = Leaf(*0*) | Node(*3*) of int*tree*tree
```

instead of

```
type tree = Leaf(*0*) | Node(*1*) of int*tree*tree
```

and similar for the declaration of the occurring list types. Now we obtain by the inference the annotated type

```
sort: 0, list_2[int,#,1|0] -> list_2[int,#,1|0], 0;
```

and we thus see that we require one additional free heap cell per node of the input list in this specific memory model. In fact, we may also notice that these additional resources are again available after the computation of `sort`, as the occurring 1 in the result type shows. Since the sorted list is expected to be of equal length to the input list, no heap memory is leaked by executing `sort` (and all subfunctions of `sort`) on *any* input list.

```
> lfd_infer heapsort -olhs 4 -ndia
This is LFD_infer  V1.12   ---   12/2003
 Program 'heapsort.lfd' parsed.
 Resource constraints constructed: 281 inequalties in 224 variables. Written to 'heapsort.constraints'.
 Solution from 'lp_solve' yields the following enriched types:
  depth         : 0, tree[0|int,#,#,0] -> int, 0;
  extract       : 0, tree[0|int,#,#,0] -> list_2[int,#,0|0], 0;
  insert        : 1, int -> tree[0|int,#,#,0] -> tree[0|int,#,#,0], 0;
  int_of_string : 0, string -> int, 0;
  make_heap     : 0, list_2[int,#,0|0] -> tree[0|int,#,#,0], 0;
  max           : 0, int -> int -> int, 0;
  print_int     : 0, int -> unit, 0;
  print_list    : 0, list_2[int,#,0|0] -> unit, 0;
  print_list_aux: 0, list_2[int,#,0|0] -> unit, 0;
  print_newline : 0, unit -> unit, 0;
  print_string  : 0, string -> unit, 0;
  print_string_newline: 0, string -> unit, 0;
  print_tabs    : 0, int -> unit, 0;
  print_tree    : 0, tree[0|int,#,#,0] -> unit, 0;
  print_tree_aux: 0, int -> tree[0|int,#,#,0] -> unit, 0;
  removesome    : 0, tree[0|int,#,#,0] -> pairoption[0|int,tree[0|int,#,#,0],0], 0;
  removetop     : 0, tree[0|int,#,#,0] -> pairoption[0|int,tree[0|int,#,#,0],0], 0;
  siftdown      : 1, int -> tree[0|int,#,#,0] -> tree[0|int,#,#,0] -> tree[0|int,#,#,0], 0;
  sort          : 0, list_2[int,#,0|0] -> list_2[int,#,0|0], 0;
  start         : 0, list_1[string,#,0|0] -> unit, 0;
  strlist2ilist : 0, list_1[string,#,0|0] -> list_2[int,#,0|0], 0;
Total processing time: 0.02 seconds.
```

There is usually more than one solution to an LP; the program simply chooses one by a default objective function. Option `-olhs 4` alters the objective function to obtain a more pleasing solution in this case; this is rarely needed, see section 3.8. Option `-ndia` simply produces a more condensed screen print of the annotated signature.

Figure 3: Example inference on heapsort.lfd

```
> lfd_infer huffman.lfd -width 50 -ndia
This is LFD_infer  V1.10*   ---   10/2003
Constraints: 327 inequalties in 261 variables.
 combine  :
  0, list_4[tree[int,int,0|int,#,#,0|0],#,0|0]
  -> tree[int,int,0|int,#,#,0|0], 0;
 count    : 0, list_3[int,#,1|0]
  -> ip_list[0|int,int,#,1], 0;
 count_aux: 2, int -> int -> list_3[int,#,1|0]
  -> ip_list[0|int,int,#,1], 0;
 huffman  : 0, ip_list[0|int,int,#,1]
  -> tree[int,int,0|int,#,#,0|0], 0;
 insert   : 2, int -> list_3[int,#,1|0]
  -> list_3[int,#,1|0], 0;
 insert_t : 1, tree[int,int,0|int,#,#,0|0]
  -> list_4[tree[int,int,0|int,#,#,0|0],#,0|0]
  -> list_4[tree[int,int,0|int,#,#,0|0],#,0|0], 0;
 maptree  : 0, ip_list[0|int,int,#,1]
  -> list_4[tree[int,int,0|int,#,#,0|0],#,0|0], 0;
 sort     : 0, list_3[int,#,1|0]
  -> list_3[int,#,1|0], 0;
 start    : 0, list_2[string,#,2|0] -> unit, 0;
 start_char: 0, list_1[char,#,1|0] -> unit, 0;
```

Note that type `tree` has three constructors: A leaf consisting of two integers, a node containing an integer and two tree elements, and a nil-like constructor. So `tree` is rather an `option tree` type.

Figure 4: Huffman tree coding – enriched signature

3.3 Huffman tree coding

We consider a program that, given a list of characters as input, computes the Huffman coding tree for that list, which might then be used for compressing this list. The program is given in A.4 and consists of two steps: First the number of occurrences of each character is counted by function `count` and saved in list of integer pairs, each pair being the code of a character and its number of occurrences in the input list. This list of pairs is then used by function `huffman` to compute the tree.

The main function `start` (which expects a list of singleton string due to Camelot/JVM only passing string arguments) preserves its input (maybe for later encoding), while the alternative main function `start_char` will destroy it. The computed enriched signature shown in figure 4 reflects this, as `start` need one extra heap cell more per list node than `start_char`. Hence the annotation for the internal nodes of the input for `start_char` is reduced by one. Recall that we are dealing with Camelot's uniform memory model, where all nodes occupy only one heap cell, and that strings are assumed to be unboxed by default.

Let us therefore focus on `start_char` from now on. In addition to destroying its input list, computing the Huffman tree seems to require an additional heap cell per list node. By the code of `sort` given in A.4 and the annotated signature shown in figure 4, we also see that `count` is merely passing this additional resource on to function `huffman`, which consumes it.

We might want to check this behavior using the sandbox execution facility of `lfd_infer`, the call and output shown in figure 5. Surprisingly, we see that none of these extra heap cells were ever used during the computation, as the

```
> lfd_infer huffman.lfd -main "start_char"
                        -arg "abcdbaadba"
-------------------------------------------------
                 2 d
            3 <
                 1 c
      6 <
            3 b
10 <
      4 a
-------------------------------------------------
     - Initial heap size                  :   10
     - Maximum heap size during evaluation:   10
     - Current heap size after evaluation :    7
```

Although the argument `"abcdbaadba"` is given as a string for convenience, the sandboxed execution translates it to the appropriate type, which is a list of characters in the case of function `start_char`.

The program then prints the calculated Huffman coding tree to the screen, with its root on the left and the leaves to the right. Each leaf contains the symbol itself together with its number of occurrences within the original input.

Figure 5: Huffman tree coding – first run

maximum number of heap cells used at a time equals the initial amount of heap required to store the input list of length 10.

So we compute the memory manually: For a list of length n, the Huffman coding tree has at most n leaves. Since a binary tree with n leaves has $n-1$ internal nodes, and all nodes of any data structure are assumed to occupy only one heap cell in this example, the result will occupy thence $2n-1$ cells. Since `huffman` is allowed to destroy its input list of length n, requiring n additional cells is indeed reasonable.

The solution to the problem here lies within function `count`, although it seems harmless as it does not alter the resource annotations: Its output list is usually shorter than its input list, since each character occurs only once in it anymore. So all the list nodes which hold a character that has already occurred before are lost, including the additionally reserved cells for this node. Alas, we cannot even hope to know whether the input will contain some characters with multiple occurrences, as it could be that we might only receive a singleton list.

So another example run shown in figure 6 proves that all resources which the inference predicted to be required were used indeed, and that the maximum heap cells used at a time equals twice the amount of the input.

One might still wonder where the one extra cell, which is not needed in the result anymore, is lost. This is due to the fact that the algorithm implicitly does an operation which is similar to calling a '`tail`'-like function. The function commonly known as `tail` simply returns the tail of its input list. Such a function should be able to return the heap cell that the head of the list occupied before the call, but it cannot, as it could be that the argument is the heap free empty list and thus no heap cell is gained.

There are two possibilities which could save the loss for a tail function: Either extending the inference to include exception handling, which would reveal the gained heap cell in a tail function's enriched type *or* to use non-empty lists

```
> lfd_infer huffman.lfd -main "start_char"
                        -arg "abcdefghij"
-------------------------------------------------
                        1 d
                  2 <
                        1 c
            4 <
                        1 b
                  2 <
                        1 a
      6 <
                  1 j
            2 <
                  1 i
10 <
                  1 h
            2 <
                  1 g
      4 <
                  1 f
            2 <
                  1 e
-------------------------------------------------
     - Initial heap size                  :   10
     - Maximum heap size during evaluation:   20
     - Current heap size after evaluation :   19
```

Please note that the initially occupied heap space by the input for the computation is the same as in figure 5.

Figure 6: Huffman tree coding – second run

as a data structure, which are already easily definable.

So computing the Huffman coding tree is in some sense a pretty bad example for our inference, as it is very likely that the function is applied to an input where each character occurs with a great multiplicity – otherwise compression would not make sense – and therefore the inferred bounds would be much too loose. However, the inference cannot do better either, as the second example run shows, although we certainly not had this special case in mind.

3.4 Flattening flatters not

We will examine another example causing loose bounds here, which is very similar to the observations made at the end of the previous section considering the computation of the Huffman coding tree. We treat it again individually here since it exposes a weak spot of our inference.

The small example considered now is the simple flattening of a list of lists into a simple list. The according code is shown in figure 7. Both functions are allowed to destroy their input. `lfd_infer` will compute the following annotated signature:

```
append : 0, chli[End(0)|Next(char,#,0)]
         -> chli[End(0)|Next(char,#,0)]
         -> chli[End(0)|Next(char,#,0)], 0;
flatten:
0,list[Nil(0)|Cons(chli[End(0)|Next(char,#,0)],#,0)]
         -> chli[End(0)|Next(char,#,0)], 0;
```

Since all resource annotations are zero, our inference was unable to detect any deallocation of heap cells, although clearly all nodes of the outer input list for `flatten` were destroyed

```
type chli   = End(*0*) | Next(*1*) of char * chli
type list   = Nil(*0*) | Cons(*1*) of chli * list
val  append : chli -> chli -> chli
val  flatten: list -> chli

let append l r = match l with  End -> r
 | Next(h,t)@_ -> let a = append t r  in Next(h,a)

let flatten l  = match l with  Nil -> End
 | Cons(h,t)@_ -> let a = flatten t   in append h a
```

Figure 7: Flattening of lists

by the destructive match. In fact, the annotated signature is the same if we turn the destructive pattern match of `flatten` into a read-only pattern match.

The problem here is again that the amount of heap cells deallocated is not linear in the size of the result of the computation: Calling `flatten` with the argument `Nil` will in fact not deallocate any heap cell, thus justifying the inferred annotated signature. Nevertheless, from a practical point of view most applications of this function will deallocate heap cells, already a call with argument `Cons(End,Nil)` does deallocate a heap cell which is leaked. The possible countermeasures are the same as stated within the previous example.

3.5 The need for destruction

We already admitted that it might be both annoying and dangerous to force the programmer to decide whether a pattern match is destructive or not. A program might crash if a destructive pattern match is not the last access to that data. We have said that an automatic inference deciding upon the pattern match modes is a topic in current research and that we are momentarily content to isolate this issue within our theoretical work in order to combine it with future research solving this particular issue; or that restricting to a linear typing discipline rids one of that problem by allowing to safely use destructive matches only.

One might also be tempted to try the opposite and set all pattern matches to read-only mode. Of course, this results in weaker bounds on the heap space consumption, as no heap cell is ever reused. However, not all programs which admit an annotated signature using destructive pattern matches still has a valid annotated signature when we turn all pattern matches into read-only matches:

```
let f l = match l with   []       -> []
                       | (h::t)@_ -> h::(f (f t))
```

The function `f` shown above, which acts as the identity function on lists, does not allocate any fresh heap cells as the inferred annotated type `f: 0,list[0|int,#,0] -> list[0|int,#,0],0` confirms. This is correct, as for each construction of a list node, another one is destroyed by the destructive pattern match immediately before.

If we would turn the destructive pattern match into read-only mode, then there is no valid enriched type for `f` anymore. Each call to `f` with an argument list of length n would then consume one heap cell plus the heap cells consumed by calling `f` twice with an argument of length $n-1$. Hence a total of $2^n - 1$ fresh heap cells would be allocated by applying `f` to a list of length n for storing intermediate results.

```
append  : 0, list[int,#,0|0] -> list[int,#,0|0]
  -> list[int,#,0|0], 0;
qsort   : 0, list[int,#,0|0] -> list[int,#,0|0], 0;
split_by: 1, int -> list[int,#,0|0]
  -> pair[list[int,#,0|0],list[int,#,0|0],0], 0;
```

Figure 8: Annotated signature for Quicksort

Our current type system however only allows us to type functions with a linear heap space consumption, hence there is no valid annotated type for `f` anymore if we change the destructive pattern match into a read-only pattern match.

3.6 Naive Quicksort fails

We will examine the well known Quicksort algorithm now. A commonly found approach to implement Quicksort is a follows: First a pivot element is chosen. Afterwards the list is filtered twice into two sublists, one containing all elements smaller and one containing all elements bigger than the pivot element. These lists are then recursively sorted and put back together with the pivot element in between them. The corresponding code is given in appendix A.6.

In short, our analysis fails to infer an annotated signature for this common functional implementation, as it is unable to recognize that the filter conditions are mutually exclusive. We do not even try to recognize such properties, as this would lead to methods of symbolic evaluation. The solution here is to rewrite the program slightly. Adjusting the program might be assisted by inspecting the output of `lfd_infer`, which is also given within appendix A.6, as it helps pinpointing the problem in the case of failure.

The idea here is to split the input list by the pivot into two sublists rather than filtering the whole list twice, so that our analysis is able to deduce that the sum of the length of both sublists is equal to the length of the list before splitting. One may also argue that this implementation is more efficient in terms of time, as it traverses its input list only once in each recursive step. This implementation of Quicksort is given in appendix A.5; its annotated signature is shown in figure 8.

Note that `qsort` does not require any extra heap memory resources although the subfunction `split_by` requires a heap cell to store the pair. `qsort` simply uses the cell previously occupied by `h` for this purpose, as the returned pair can be destructed before the list node for `h` has to be reconstructed again.

One may also argue that this use of pairs or tuples for merely passing multiple results or arguments might be implemented using stack space only. We might reflect this if we want to by changing the number of heap cells needed to construct an element of type `pair` to zero within the type declaration for `pair`.

3.7 Sharing and subtyping

`lfd_infer` allows subtyping with respect for annotations. Although subtyping seems alway to imply a memory leakage, it enables us to infer solutions where there would be none otherwise. Consider the following enriched types:

```
l: list[int,#,6|0]
f: 0, list[int,#,1|0] -> list[int,#,0|0], 0;
g: 0, list[int,#,5|0] -> list[int,#,5|0], 0;
```

Annotation subtyping enables us to form either the expression `f(l)` or `g(l)` (but not both simultaneously), which would not be well typed otherwise. But even if both functions `f` and `g` would not modify their input (read-only access only), and and therfore would not change the length of `l` during the computation, we would leak memory with both calls, as the type of the result is determined by the functions type. Hence we cannot type an expression like `g(f(l))`.

The solution to this problem lies within *sharing*. When a variable is shared or aliased, the resource annotation within its type are distributed linearly, i.e. since $6 \geq 1 + 5$ we could share or alias `l` to `l1: list[int,#,1|0]` and `l2: list[int,#,5|0]`. Now we might form both calls `f(l1)` and `g(l2)` without leaking any memory, we could even instead share `l2` again to call `f` another time. Note that sharing is done always implicitly, the names `l1` and `l2` were only used for didactical reasons here.

Yet there is another problem as already pointed out in section 1.1: If we want to form both calls, we must be sure that the first call does not destroy the list and creates a dangling pointer. So we have to look up the definition of `f` and `g` and see whether one of them might use a destructive match on their input. Note that in our system a node of a data structure on the heap can only be destroyed and built anew, but it cannot be updated.[‡]

A program that might not have a valid enriched type might become typeable by using subtyping. This would result in memory leak, which could be detected by the methods described in section 3.8 and would point us to the place in the code where we might then be able to make good use of sharing to avoid it.

We must admit that there is another problem with sharing as it is implemented now: Consider the code fragment `match l with Cons(h,t) -> if b then g(l) else g(t)` where `l` is shared. This fragment is not typeable under the assumption on the types of `l` and `g` as stated above, as either `l: list[int,#,5|0]` and `t: list[int,#,0|0]` *or* `l: list[int,#,0|0]` and `t: list[int,#,5|0]` would be required by sharing, so we must write `g(Cons(h,t))` instead of `g(l)` which then does not involve sharing at all. This is rather annoying, but it does currently not seem us to be a crucial problem. However, among other things, this particular issue was treated in [13] in a related setting.

3.8 Format of the generated LP

Shown in figure 9 is an excerpt of the generated LP for the Insertionsort program example A.3, which was derived automatically by Camelot from code exactly as given in figure 1. This file contains exactly what is fed to the LP solver. It might probably be used with other LP solvers as well.

Apart from some statistics, the file starts with a comment containing the enriched signature of the program with variables instead of values. This is merely for convenience in order to identify to what a certain variable relates to. Note also that the first letter of each variable designates its use,

[‡]We are only interested in the overall balance of used heap cells and therefore abstract away from heap addresses. We still speak of an in-place update of e.g. a list in a sorting algorithm, if for each list node allocated another node was deallocated before. Hence the number of live heap cells never exceeded the initial value during the whole computation.

However, Camelot does allows allocation at specific heap locations, but we regard this rather as a runtime optimization, since it does not affect the number of live heap cells at any time.

```
/*This file is an automatically generated lp.
  Contains 26 inequalties in 25 variables.
*//*
insert: <x02>, int
  -> list_1[Cons(int,#,<u01>)|Nil(<u02>)]
  -> list_1[Cons(int,#,<v01>)|Nil(<v02>)], <y02>;
sort  : <x03>, list_1[Cons(int,#,<u03>)|Nil(<u04>)]
  -> list_1[Cons(int,#,<v03>)|Nil(<v04>)], <y03>;
*/
MIN:  +4*u01 +4*u02 +4*u03 +4*u04 -2*v01 -2*v02
      -2*v03 -2*v04 +2*x02 +2*x03 -1*y02 -1*y03 ;
insert_10_M'2: +1*a04 -1*u02 -1*x02        <= 0 ;
insert_Sub:    -1*c01 +1*v01               <= 0 ;
insert_Sub:    -1*c02 +1*v02               <= 0 ;
insert_12_Con: +1*K1  -1*a05 +1*v01 +1*y02 <= 0 ;
insert_11_Con: +0*K1  -1*a04 +1*a05 +1*c02 <= 0 ;
insert_13_Ma1: -1*K1  +1*a06 -1*u01 -1*x02 <= 0 ;
insert_20_App: -1*a06 +1*a07 +1*x02 -1*y02 <= 0 ;
insert_20_App: -1*a06 +1*x02               <= 0 ;
insert_21_Con: +1*K1  -1*a07 +1*v01 +1*y02 <= 0 ;
insert_16_Con: +1*K1  -1*a06 +1*a08 +1*c03 <= 0 ;
insert_Sub:    +1*c04 -1*u02               <= 0 ;
insert_Sub:    +1*c03 -1*u01               <= 0 ;
insert_17_Con: +1*K1  -1*a08 +1*v01 +1*y02 <= 0 ;
insert_Sub:    -1*c04 +1*v02               <= 0 ;
insert_Sub:    -1*c03 +1*v01               <= 0 ;
```

This excerpt of the LP generated by `lfd_infer` for the Insertionsort program given in appendix A.3 shows header, objective function and all constraints generated for function `insert`. For reasons of space we altered whitespace and deleted all 11 inequalities for function `sort`, as well as all 50 constraints ensuring non-negativity and an upper bound for each variable.

Figure 9: Generated LP for Insertionsort

e.g. `K3` stands for the constant value 3, `x02` stands for the second functions fixed input of additional resources, etc.
Afterwards an objective function is printed, using only variables that appear within the signature. As stated earlier, the objective function tells the LP solver how to choose a solution if more than one solution to the LP is possible, i.e. if there are several valid enriched signatures. It is not entirely clear what the objective function should be, e.g. the type `0, list_1[int,#,1|0] -> list_1[int,#,0|0], 0;` might be preferable in some cases, while in other cases the type `99, list_1[int,#,1|0] -> list_1[int,#,1|0], 0;` would be better, depending on the expected average length of the result. (A small, but artifical example admitting both annotated types can be constructed.) The factors for each kind of variable within the objective function can thus simply be altered by options.

The important question is whether there exists a solution at all, as grouping individual functions together to a program might severely restrict the number of valid enriched types for each function individually, although subtyping might help here as well.

We also include upper bounds for each variable as well as non-negativity constraints, in order to always obtain a solution (and hence an enriched typing) if the set of solutions is not only infinite, but the objective function would also admit an infinite value. The Heapsort example is such a case for the default objective function.

After the objective function, we see the generated constraints. Each inequality bears a label, which tells us which function was responsible for generating this constraint (except for upper and lower bounds, all constraints are due to a specific function), the line number of the responsible code, and finally which rule was applied. These rule tags directly correspond to the typing rules given in [7], and they are described within the manual of `lfd_infer`.

The program listing A.3 was printed with explicit line numbers in order to facilitate considering an example:
`insert_13_Ma1: -1*K1 +1*a06 -1*u01 -1*x02 <= 0;`
According to its tag, the inequality was generated when examining function `insert`, line 13. The tag `Ma1` tells us further that this is a branch of a destructive pattern match on the first constructor of the matched type (i.e. a `Cons`). Since all variables are restricted to non-negative values, the sum of variables with negative factors represent the amount of available heap cells before executing the corresponding operation; the sum of variables having a positive factor represents the amount of heap cells available for all paths of computation trailing this operations.

Hence in this example, we have `x02` heap cells available guaranteed by the caller of `insert`, `u01` unused heap cells due to the successful match of a `Cons` on the input, and finally one heap cell since the match was destructive (variable `K1` is always restricted to 1). All continuing computations following this point will use at most `a06` heap cells. `a06` is an intermediate variable that will be used from this point on instead of `x02`, as we can see on all the inequalities which immediately follow this point of computation, i.e. lines 16 and 20, which contain `a06` negatively now. Please compare this to the considerations we did in section 1.

An inequality which is strict for a fixed solution indicates a memory leak within the corresponding branch of computation of that function under the enriched typing derived from that solution. Tracing this information back to the source code is clearly possible and desirable, but is just not implemented yet.

4. CONCLUSION

The implementation `lfd_infer` of the heap space inference presented by M. Hofmann and S. Jost shows that this inference is indeed efficiently usable on a wide range of programs. Compared to approaches based on symbolic evaluation like [17], our system reaches a much higher efficency as shown in section 3.1, but of course we trade in the range of covered programs and the accuracy of the results obtained for achieving this.

We already mentioned several points for improvement, e.g. tracing back slack of the LP to the original program code to expose memory leakage, as commented on in section 3.8.

Many readers also suggest that statically determining the pattern match mode, as stated in section 1.1, in order to avoid dangling pointers is highly desirable; or that one should care about non-termination as well, since these issues are all related safety properties of programs. While this is true, it might be the case that the problem is far to complex to be covered within a single analysis method. Furthermore, even if a certain program run might fail or loop, the inferred bounds on heap space consumption will still hold. Therefore we are currently content to leave these problems to research specifically dedicated to such issues, and insulate the crucial

properties required in order to combine all efforts later on. A factor for the tightness of the bounds inferred by our method is whether a program traverses its data structures entirely before deallocating them, and that their sizes play a role in the ongoing computation as stated in section 3.4. If this is not the case, our inference ignores the benefits which could be obtained for example from garbage collection. Of course, garbage collection could be used to increase the size of the freelist during the computation, but the reclaimed heap cells would be useless: If we want to be sure that the program will not fail due to a lack of heap memory, we must provide as many heap cells as our inference had told us already right from the start of the execution. It is in the nature of a static analysis that it does not benefit from observations which may only be performed at runtime. Nevertheless, bounds on heap memory usage established by our inference are *guaranteed* to hold, although the slack compared to the actually required heap space in the presence of garbage collection might be great.

We currently investigate whether some of the methods employed by region based memory management [16, 14, 5] or sized types [8] could help us to improve the tightness of the derived bounds. Furthermore the combined methods might enable us to extend the inference to true higher-order programs (the lifetimes of closures might be the main problem here) or polymorphism with respect to resource annotations.

A further discussion of related scientific work is already contained in [7].

5. ACKNOWLEDGMENTS

This work was supported by the Graduiertenkolleg Logik in der Informatik (DFG). The author thanks his supervisor Martin Hofmann for discussing this work. I also thank Martin Elsman and Henning Niss for hosting me at the IT University of Copenhagen (where I wrote parts of this work) and for some fruitful discussions concerned with the future of the underlying theoretical work. Thanks to some of the anonymous referees for their valuable remarks.

6. REFERENCES

[1] Mobile resource guarantees. EU Project No. IST-2001-33149, see `http://www.dcs.ed.ac.uk/home/mrg/`.

[2] Objective caml. An open source compiler. Version used is 3.06 `http://caml.inria.fr/`.

[3] D. Aspinall and M. Hofmann. Another type system for in-place update. In *Proc. 11th European Symposium on Programming, Grenoble*, volume 2305 of *Lecture Notes in Computer Science*. Springer, 2002.

[4] M. Berkelaar. lp_solve. A LP-solver released under the Lesser GNU public licence. Version used is 4.0.1.0. Eindhoven University of Technology `ftp://ftp.es.ele.tue.nl/pub/lp_solve`.

[5] F. Henglein, H. Makholm, and H. Niss. A direct approach to control-flow sensitive region-based memory management. In *Proceedings of the 3rd International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP)*, pages 175–186, Montréal, Canada, 2001. ACM.

[6] M. Hofmann. A type system for bounded space and functional in-place update. *Nordic Journal of Computing*, 7(4):258–289, Autumn 2000. An earlier version appeared in ESOP2000.

[7] M. Hofmann and S. Jost. Static prediction of heap space usage for first-order functional programs. In *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 185–197. ACM, 2003.

[8] J. Hughes and L. Pareto. Recursion and dynamic data structures in bounded space: towards embedded ML programming. In *Proc. International Conference on Functional Programming (ACM). Paris, September '99.*, pages 70–81, 1999.

[9] S. Jost. The implementation `lfd_infer` of the static inference on heap space usage is available at `www.tcs.informatik.uni-muenchen.de/~jost`.

[10] S. Jost. Static prediction of dynamic space usage of linear functional programs, 2002. Diploma thesis at Darmstadt University of Technology, Department of Mathem atics. Available at `www.tcs.informatik.uni-muenchen.de/~jost/da_sj_28-02-2002.ps`.

[11] N. Kobayashi. Quasi-linear types. In *Proceedings ACM Principles of Programming Languages*, pages 29–42, 1999.

[12] M. Konečný. Functional in-place update with layered datatype sharing. In *TLCA 2003, Valencia, Spain, Proceedings*, pages 195–210. Springer-Verlag, 2003. Lecture Notes in Computer Science 2701.

[13] O. Lee, H. Yang, and K. Yi. Inserting safe memory reuse commands into ml-like programs. In *Proceedings of the Annual International Static Analysis Symposium*, volume 2694 of *Lecture Notes in Computer Science*, pages 171–188, San Diego, California, June 2003. Springer-Verlag.

[14] M. Tofte, L. Birkedal, M. Elsman, N. Hallenberg, T.H. Olesen, P. Sestoft. Programming with regions in the ml kit, April 2002. IT University of Copenhagen `http://www.itu.dk/research/mlkit/`.

[15] M. Odersky. Observers for linear types. In B. Krieg-Brückner, editor, *ESOP '92: 4th European Symposium on Programming, Rennes, France, Proceedings*, pages 390–407. Springer-Verlag, February 1992. Lecture Notes in Computer Science 582.

[16] M. Tofte and J.-P. Talpin. Region-based memory management. *Information and Computation*, 132(2):109–176, 1997.

[17] L. Unnikrishnan, S. D. Stoller, and Y. A. Liu. Automatic accurate live memory analysis for garbage-collected languages. In *Proceedings of The Workshop on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, pages 102–111. ACM, 2001.

[18] K. Wansbrough and S. Jones. Simple usage polymorphism, 2000. ACM SIGPLAN Workshop on Types in Compilation, 2000. To appear.

[19] N. Wolverson and K. MacKenzie. Camelot and Grail: Compiling a resource-aware functional language for the java virtual machine. Developed during EU Project No. IST-2001-33149. `http://www.dcs.ed.ac.uk/home/mrg/publications/`.

APPENDIX

A. PROGRAM EXAMPLE FILES

All the covered examples are included in the `lfd_infer` package available at [9]. The examples treated here are:

A.1 big100.lfd

This is an excerpt only. The full code is lengthy and highly redundant. It should be easy to reconstruct.

Please note that although this program is pretty artifical and easy to analyze from a human programmer's point of view, this is not the case for our automatic inference, as it lacks a bird's eye view and thus the recognition of the repeated pattern. It does not even try to: The analysis of each function of a program is entirely independent. Nevertheless, the interweaved function calls ensure that the generated LP cannot be split into independent blocks and thus must be solved as a non-trivial whole.

```
type list = Nil(*0*) | Cons(*2*) of int * list

val main : string -> list
val id_000 : list -> list
val id_001 : list -> list
  .
  .
val id_098 : list -> list
val id_099 : list -> list

let main s =
 let l = (* Build some list depending on s *)
  .
  .
 in id_000 l

let id_000 l =
  match l with
  | Nil@d        => let _ = print "."
                    in Nil@d
  | Cons(h, t)@d => let t'  = id_099 t   in
                    let t'' = id_000 t'  in
                    let c   = Cons(h,t'')@d in
                    c


let id_001 l =
  match l with
  | Nil@d        => Nil@d
  | Cons(h, t)@d => let t'  = id_000 t   in
                    let t'' = id_001 t'  in
                    let c   = Cons(h,t'')@d in
                    id_000 c
  .
  .
let id_099 l =
  match l with
  | Nil@d        => Nil@d
  | Cons(h, t)@d => let t'  = id_098 t   in
                    let t'' = id_099 t'  in
                    let c   = Cons(h,t'')@d in
                    id_098 c
```

A.2 heapsort.cmlt

This program is written for Camelot and makes use of Camelot's built-in lists. In order to apply the inference to it, one must preprocess it with the Camelot compiler using option `-a2`. This listing is complete.

```
type tree       = !Leaf | Node of int * tree * tree
type pairoption = !None | Some of int * tree

let max x y = if x > y then x else y

let depth t = match t with  Leaf -> 0
    | Node(v,l,r) -> 1 + max (depth l) (depth r)

let strlist2ilist l = match l with  [] -> []
    | (y::t)@_ -> int_of_string y :: strlist2ilist t
```

Some simple printing functions...

```
let print_list_aux l = match l with [] -> ()
    | y::t -> let _ = print_string ", " in
              let _ = print_int y       in
              print_list_aux t

let print_list l = let _ = print_string "[" in
    match l with [] -> print_string_newline "]"
    | y::t -> let _ = print_int y      in
              let _ = print_list_aux t in
              print_string_newline "]"

let print_tabs i = if i>0 then
      let a = print_string " " in (print_tabs(i-1))
      else ()

let print_tree_aux i t = let stretch = 6 in
    match t with
      Leaf -> let _ = print_tabs(i*stretch)    in
              let _ = print_string_newline "L" in ()
    | Node(v,l,r) ->
        let _ = print_tree_aux (i+1) r    in
        let _ = print_tabs(i*stretch)     in
        let _ = print_int(v)              in
        let _ = print_string_newline " <" in
        let _ = print_tree_aux(i+1)l      in ()

let print_tree t = let _ = print_newline()    in
                   let _ = print_tree_aux 0 t in
                   let _ = print_newline()    in ()
```

All trees are supposed to be heaps. `Leaf` is a heap and `t=Node(w,t1,t2)` is a heap, if `w` is the largest element of `t` and moreover `t1`, `t2` are heaps and $0 \leq |\texttt{t1}| - |\texttt{t2}| \leq 1$.

```
let insert x t = match t with
      Leaf                -> Node(x,Leaf,Leaf)
    | Node(y,left,right)@_ -> if x<=y
       then Node(y, insert x right, left)
       else Node(x, insert y right, left)
```

Function `siftdown w t1 t2` assumes that `t1`, `t2` are heaps and that $0 \leq |\texttt{t1}| - |\texttt{t2}| \leq 1$. It returns a heap consisting of the elements of `Node(w,t1,t2)`.

```
let siftdown w t1 t2 = match t1 with
  Leaf              -> Node(w,Leaf,Leaf)
```

```
| Node(v,t11,t12)@_ -> (match t2 with
    Leaf -> if w>=v
      then Node(w,Node(v,Leaf,Leaf),Leaf)
      else Node(v,Node(w,Leaf,Leaf),Leaf)
  | Node(u,t21,t22)@_ -> if w>=u & w>=v
   then Node(w,Node(v,t11,t12),Node(u,t21,t22))
   else if u>=w & u>=v
    then Node(u,Node(v,t11,t12),siftdown w t21 t22)
    else Node(v,siftdown w t11 t12,Node(u,t21,t22)))
```

Function `removesome t` removes an arbitrary element from `t` and returns it as well as the resulting heap.

```
let removesome t = match t with  Leaf -> None
  | Node(x,left,right)@_ ->
    (match removesome left with None -> Some(x,right)
                  (* actually, right=Leaf, here *)
     | Some(y,left')@_
        ->  Some(y,Node(x,right,left')))
```

Function `removetop t` removes the largest element from a heap.

```
let removetop t = let x = print_tree t in
  match t with  Leaf -> None
  |  Node(x,left,right)@_ ->
     (match (removesome left) with
        None     ->  Some(x,right)
                     (* actually, right=Leaf here *)
      | Some(y,left')@_
                ->  Some(x,siftdown y right left'))

let extract h = match removetop h with  None -> []
  | Some(h,t)@_ -> h :: extract t

let make_heap l = match l with  [] -> Leaf
  | (h::t)@_ -> let hp = make_heap t   in
                let hc = print_tree hp in
                insert h hp

let sort l = extract (make_heap l)

let start arg = print_list(sort(strlist2ilist arg))
```

A.3 insertionsort.lfd

This is the Insertsort program shown in figure 1, ready to be fed to `lfd_infer`.

We obtained this code by preprocessing the code shown in figure 1, using Camelot with option `-a2`. We edited whitespace and added line numbers manually, for the readers convenience. Also for readability, we changed the constructor names `Cons$_1` and `Nil$_1` to `Cons` and `Nil` respectively, which Camelot uses for its built-in lists to avoid name clashes.

```
 1: type  list_1 = Cons(*1*) of int *list_1
 2:
 3:  | Nil(*0*)
 4: val main: (string array -> unit)
 5: val insert: (int -> (list_1 -> list_1))
 6: val sort: (list_1 -> list_1)
 7:
 8: let insert e l =
 9: begin match l with
10:     Nil ->
11:       let ?t0 = Nil in
12:       Cons(e, ?t0)
13:   | Cons(h,t)@_ ->
14:       if (e < h)
15:       then
16:         let ?t1 = Cons(h, t)
17:         in Cons(e, ?t1)
18:       else
19:         begin
20:           let ?t2 = insert (e) (t)
21:           in Cons(h, ?t2)
22:         end
23: end
24:
25: and sort l =
26: begin
27:   match l with
28:     Nil ->
29:       Nil
30:   | Cons(h,t) ->
31:       let ?t3 = sort (t) in
32:       insert (h) (?t3)
33: end
```

A.4 huffman.cmlt

This program computes the Huffman coding tree for a given character list (or a list of singleton strings for a reason of compatibility with the JVM).

Printed here is an excerpt of the essential functions: a screen printing function `print_tree` for type `tree` and a sorting function `sort` on integer lists have been omitted in this listing, as such functions are already contained in the listings A.2 and A.3 respectively. The tree type in A.2 differs slightly from the tree type used here, but adjusting `print_tree` should be obvious.

```
type tree    = Leaf of int * int
             | Node of int * tree * tree
             | !Error
type ip_list = !Nil | Cons of int * int * ip_list

let count_aux i c l =
  match l with [] -> Cons(i, c, Nil)
  | (h::t)@_ -> if h = i then count_aux i (c+1) t
                else Cons(i, c, (count_aux h 1 t))
let count l =
  match sort l with [] -> Nil
  | (h::t)@_           -> count_aux h 1 t

let getkey t =  match t with
  Leaf(i,s) -> i | Node(i,l,r) -> i | Error -> -1

let insert_t t1 l =
 match l with [] -> t1::[]
 | (t2::ls)@_ ->
   let k1 = getkey(t1) in let k2 = getkey(t2) in
   if k2>k1 then t1 :: t2 :: ls
            else t2 :: insert_t t1 ls

let maptree l =
  match l with Nil@_   -> []
  | Cons(sym,key,tl)@_ ->
     insert_t (Leaf(key,sym)) (maptree tl)
```

```
let combine l =
 match l with [] -> Error (* Shouldn't happen *)
 | (t1::ls)@_    ->
   (match ls with [] -> t1 (*Wastes a resource*)
    | (t2::lss)@_    -> let k1 = getkey(t1) in
       let k2 = getkey(t2) in
       combine(insert_t(Node(k1+k2,t1,t2)) lss))

let huffman l  = combine(maptree l)

let intlist_of_stringlist l =     (* Copies input *)
  match l with [] -> []  |  y::t ->
    int_of_string y :: intlist_of_stringlist t

let intlist_of_charlist l =     (* Destroys input *)
  match l with [] -> []  |  (y::t)@_ ->
    int_of_char y :: intlist_of_charlist t

let start args =
  let l = intlist_of_stringlist args in
  print_tree(huffman(count(l)))

let start_char args =
  let l = intlist_of_charlist args in
  print_tree(huffman(count(l)))
```

A.5 qsort.cmlt

This is an implementation of the well known Quicksort algorithm. The function `qsort` is allowed to destroy its input and sorts the list in-place without allocating any extra resources. The annotated signature was given in figure 8.

```
type 'a pair = Pair(*1*) of 'a * 'a

let append l r =
  match l with [] -> r
  | (h::t)@_ -> h::(append t r)

let split_by p l =
  match l with [] -> Pair([],[])
  | (h::t)@_ ->
      match split_by p t with
        Pair(s,b)@_ -> if (h < p) then Pair(h::s,b)
                                  else Pair(s,h::b)
let qsort l =
  match l with [] -> []
  | (h::t)@_ ->
      match split_by h t with
        Pair(s,b)@_ ->
          append (qsort s) (h::(qsort b))
```

A.6 Naive Quicksort

This program is a common implementation of the Quicksort algorithm. Contrary to the code given in appendix A.5, which also implements the Quicksort algorithm, `lfd_infer` will fail when applied to this code. We included this code to exhibit a problematic example for our analysis and we discuss the issue more closely in section 3.6.

```
let xor a b = (a || b) && (not (a && b))

let append l r = match l with [] -> r
                 | (h::t)@_ -> h::(append t r)

let filter b p l =
  match l with []   -> []
             | h::t -> if xor b (p < h)
                       then h::(filter b p t)
                       else (filter b p t)
let qsort l =
  match l with [] -> []
  | (h::t)@_ -> append (qsort (filter true  h t))
                       (h::(qsort (filter false h t)))
```

The code above is written in the language of Camelot. If `lfd_infer` is applied to the code as transformed by the Camelot compiler, it fails and reports the following:

```
> lfd_infer naive_qsort -ndia
 Resource constraints constructed:
   60 inequalties in 48 variables.
 Calling 'lp_solve' via pipe...

 --- LP FOR THE WHOLE PROGRAM IS INFEASIBLE !!! ---

 Trying to solve constraints for each defined
 function individually.

 NOTE:  while some functions may produce feasible
 LPs on their own, plugging these functions together
 into a program may lead to an infeasible LP as well

append: 0,list[int,#,0|0] -> list[int,#,0|0]
        -> list[int,#,0|0],0;
filter: 0,bool -> int -> list[int,#,0|0]
        -> list[int,#,9999|0],0;
qsort : 0,list[int,#,0|0] -> list[int,#,10000|0],0;
xor   : 0,bool -> bool -> bool, 0;
```

The value 10000 is the default upper bound for all variables, i.e. that means that it might be replaced by an arbitrarily high value. Of course, except for some odd examples like `let f () = []` which may indeed have the valid annotated type `f: 0, unit -> list[int,#,10000|0],0`, this cannot be. No function may deallocate an arbitrary number of heap cells regardless of its input. So we have to inspect the functions `qsort` and `filter` more closely.

Since the inference failed for the whole program, each function is examined separately to help the programmer to identify the problematic function. So the function `filter` may gain an arbitrary number of resources simply by calling `xor` as a subfunction: During the analysis of `filter` the annotated type of `xor` is treated independently from the analysis of `xor` itself. The same applies to `qsort` which calls `filter`. This behavior may seem odd, but it is the case that the inference may already fail for a particular function regardless of the annotated types of the called subfunctions.

We investigate further by replacing the call to `xor` within `filter` by inlining, which we did not in the first place for an easier understanding of the code. Now the inference will still fail, but it now reports the following annotated type for `filter`:

```
filter: 0,bool -> int -> list[int,#,1|0]
        -> list[int,#,0|0],0;
```

This is the correct type now, because `filter` is not allowed to destroy its input, it must build the result within fresh

heap space. Since each element of the input might pass the filter, the function requires an extra heap cell for each internal list nodes of the input list. Note that this extra heap resource is leaked if the element does not satisfy the filter criterion.

Now we know that the inference for `qsort` must fail: Since it calls filter twice on its input list, the annotated type for the input list should be `list[int,#,2|0]`. But the applying `filter` will return a list of type `list[int,#,0|0]` in both cases, so we cannot call `qsort` recursively on these parts.

Furthermore the intermediate lists constructed by `filter` are never deallocated, except for the first element of each list. This defect could be solved by replacing the second call with a destructive version of `filter`. Admittingly this would be good reason to include an analysis for automating the pattern match mode to avoid code duplication. However, even this would not help here: The first call would still leak all the extra resources for the filtered elements. In addition, all filtered elements in the second destructive application of `filter` are leaked as well. Thus in each recursive step a number of heap cells equal to the length of the partial list are leaked, leading to a super-linear consumption of heap cells.

A.7 mergesort.lfd

This is an implementation of the Mergesort algorithm. The inferred signature is included after the program code.

```
type 'a pair = Pair of 'a * 'a

let merge l r = match l with [] -> r
  | (h1::t1)@_ -> begin
      match r with
        [] -> (h1::t1)  (* l is not allowed here *)
      | (h2::t2)@_ ->  if h1 < h2
                       then h1::(merge t1 (h2::t2))
                       else h2::(merge (h1::t1) t2)
      end

let split l xs ys = match l with [] -> Pair(xs,ys)
  | (h1::t1)@_ -> begin
      match t1 with [] -> Pair(h1::xs,ys)
      | (h2::t2)@_ -> split t2 (h1::xs) (h2::ys)
      end

let msort l = match l with []   -> []
  | h::t ->
     match (split l [] []) with Pair(l,r)@_ ->
       merge (msort l) (msort r)
```

There is only a minor problem related to variable sharing in this example, namely that we cannot conveniently write `l` instead of `h1::t1` at the commented place within function `merge`, although both terms are semantically equivalent. We already commented on this problem in section 3.7.

Our inference `lfd_infer` will respond the following when applied to the above program:

```
> lfd_infer merge_sort_cmlt -ndia
This is LFD_infer  V1.13   ---   12/2003
 Program 'merge_sort_cmlt.lfd' parsed.
 Resource constraints constructed:
   87 inequalties in 69 variables.
 Solution from 'lp_solve' yields:

merge: 0, list[int,#,0|0] -> list[int,#,0|0]
       -> list[int,#,0|0], 0;
msort: 1, list[int,#,0|0] -> list[int,#,0|0], 1;
split: 1, list[int,#,0|0] -> list[int,#,0|0]
       -> list[int,#,0|0]
       -> pair[list[int,#,0|0],list[int,#,0|0],0],0;
```

Thus an application of function `msort` to an arbitrary list requires an additional heap cell. This extra heap cell is again available after the computation, thus it was only needed to store an intermediate result. In fact, this heap cell was required to hold the pairs returned by each call to function `split`.

B. TECHNICAL DETAILS

The constructed LP is stored in a standard and human-readable format for LPs. By default, the inference calls the LP-solver `lp_solve` [4], which is released under the lesser GNU public license and is assumed to be accessible from the current path, to solve the constructed LP.

`lfd_infer` was implemented in Objective Caml [2].

B.1 Built-in types

Type	Constructors	Operations
`unit`	`()`	
`diamond`		
`<>`		
`bool`	`true false`	`and && or || not`
`int`	`0 1 -2 ...`	`+ - * / mod`
`float`	`0 1.2 -3.4e-5 ...`	`+. -. *. /.`
`char`	`' ' 'a' 'B' ...`	
`string`	`"Hello" ...`	`^`

LFD knows the types listed above, the last five being directly treated by OCaml. All built-in types are treated as heap free, although this might be unreasonable in some situations, e.g. if strings are used for more than just on screen printing, they should be replaced by user defined type such as a list of characters.

The type `diamond` is treated as the unit type, but has no constructor; `<>` is simply an alias for it. It is there for compatibility with Camelot.

The append operator `^` accepts as its arguments both type `string` and `char` and returns always an element of type `string`.

Furthermore the operations `=`, `>`, `<`, `<=` and `>=` operate on most of these types and do the expected.